**Introduction to Java**

Java is a high-level, object-oriented programming language developed by Sun Microsystems in 1995. It has since been acquired by Oracle Corporation. Java is designed to be platform-independent, meaning that programs written in Java can run on any system that has a compatible Java environment, regardless of the underlying hardware or operating system.

Java's slogan, "Write Once, Run Anywhere" (WORA), highlights this key feature, thanks to its platform-independent bytecode that is interpreted and executed by the Java Virtual Machine (JVM). Java is widely used in various domains, such as:

* **Web applications** (e.g., server-side applications using Java EE)
* **Mobile applications** (e.g., Android development)
* **Enterprise applications** (e.g., banking systems, ERP)
* **Scientific computing**
* **Big Data processing**

**Key Features of Java:**

* **Object-Oriented:** Everything in Java is an object, which means it follows principles like inheritance, encapsulation, polymorphism, and abstraction.
* **Platform-Independent:** Through the use of the JVM, Java programs can run on any platform with a compatible JVM implementation.
* **Multithreading:** Java provides built-in support for multithreading, allowing the execution of multiple tasks concurrently.
* **Rich API:** Java has a rich set of libraries that cover everything from networking and database connectivity to security and graphical user interfaces (GUIs).

**Java Versions**

| **Version** | **Release Year** | **Important Features** |
| --- | --- | --- |
| Java 1.0 | 1996 | First stable release, basic OOP features |
| Java 1.1 | 1997 | Introduced inner classes, JDBC API, RMI |
| Java 1.2 | 1998 | Swing GUI, Collections framework, JIT compiler |
| Java 1.3 | 2000 | Performance improvements, Java Sound API |
| Java 1.4 | 2002 | Assertions, NIO (New I/O), XML processing |
| Java 5 (1.5) | 2004 | Generics, Metadata, Enhanced for-loop |
| Java 6 | 2006 | Scripting API, Performance enhancements |
| Java 7 | 2011 | Try-with-resources, Fork/Join framework |
| Java 8 | 2014 | Lambda expressions, Streams API, Time API |
| Java 9 | 2017 | Module system, JShell REPL |
| Java 10 | 2018 | Local-variable type inference (var) |
| Java 11 | 2018 | Long-term support (LTS), New HTTP client |
| Java 12 | 2019 | Switch expressions (preview), Shenandoah GC |
| Java 13 | 2019 | Text blocks (preview), Dynamic CDS archives |
| Java 14 | 2020 | Records (preview), NVM support |
| Java 15 | 2020 | Sealed classes (preview), Hidden classes |
| Java 16 | 2021 | Records, Pattern matching (preview) |
| Java 17 | 2021 | LTS release, Strong encapsulation |
| Java 18 | 2022 | Code snippets in Javadoc, Simple Web Server |
| Java 19 | 2022 | Virtual Threads (preview), Record Patterns (preview) |
| Java 20 | 2023 | Scoped Values, Foreign function |
| Java 21 | 2023 | String Template |
| Java 22 | 2024 | Stream Gatherers, Structured Concurrency |
| Java 23 | 2024 | Primitive Type Patterns, Flexible Constructor bodies |

**Java Platform Independence**

Java achieves platform independence through the use of the Java Virtual Machine (JVM). When a Java program is compiled, it is not compiled into native machine code but into an intermediate form known as bytecode. This bytecode is executed by the JVM, which abstracts the underlying hardware and operating system. As a result, Java programs can run on any platform that has a compatible JVM, ensuring that the code can be executed anywhere without modification.

**JDK, JRE & JVM**

**Java Development Kit (JDK)**

The **Java Development Kit (JDK)** is a software development kit used to develop Java applications. It provides all the tools necessary to write, compile, debug, and execute Java programs. The JDK includes the following components:

**Components of the JDK:**

1. **JRE (Java Runtime Environment):** This is the runtime environment for executing Java programs. The JRE includes the JVM, the standard Java class libraries, and other resources that allow Java programs to run.
2. **Java Compiler (javac):** This tool compiles Java source code into bytecode (.class files) that can be executed by the JVM.
3. **Java Debugger (jdb):** A tool used for debugging Java programs by allowing developers to inspect and manipulate the state of a program while it is running.
4. **Java Documentation Generator (javadoc):** This tool generates HTML documentation from Java source code based on special comments written in the code (Javadoc comments).
5. **Other Development Tools:** These include various utilities for packaging, deployment, and testing, such as jar for creating Java archives, and java for running compiled Java programs.

The JDK is essential for developers because it contains everything they need to create Java applications from scratch, including writing, compiling, and debugging code.

**Java Runtime Environment (JRE)**

The **Java Runtime Environment (JRE)** is a part of the JDK and represents the environment in which Java programs run. The JRE includes the Java Virtual Machine (JVM), core libraries, and other components necessary for the execution of Java programs. However, it **does not** include the development tools needed to create Java applications (like compilers or debuggers).

**Key Components of JRE:**

* **JVM (Java Virtual Machine):** The JVM is responsible for interpreting and running Java bytecode.
* **Core Libraries:** These are the essential class libraries that Java programs need to function, such as classes for input/output, networking, data structures, utilities, etc.
* **Other Runtime Components:** These include support for running applications on the underlying operating system, file handling, network communication, etc.

The JRE is sufficient for running precompiled Java applications but does not contain the tools required to develop new Java applications.

**Java Virtual Machine (JVM)**

The **Java Virtual Machine (JVM)** is the engine that provides runtime support for executing Java bytecode. It plays a critical role in the WORA principle, enabling Java applications to be platform-independent. The JVM takes compiled bytecode (.class files) and interprets or compiles it into machine code that is understood by the underlying operating system and hardware.

**Key Responsibilities of the JVM:**

1. **Loading Code:** The JVM loads compiled Java bytecode from .class files into memory.
2. **Verifying Code:** Before execution, the JVM checks the bytecode for security and validity.
3. **Executing Code:** The JVM executes bytecode instructions either through interpretation or Just-In-Time (JIT) compilation, depending on the JVM implementation.
4. **Memory Management:** The JVM is responsible for allocating and managing memory for Java applications, including heap and stack memory management.
5. **Garbage Collection:** The JVM automates memory management by reclaiming memory used by objects that are no longer needed.

**JVM Architecture:**

* **Class Loader Subsystem:** Loads classes into memory.
* **Runtime Data Areas:** Includes the heap (for storing objects), method area (for storing class structures), stack (for method calls), and program counter (PC) register.
* **Execution Engine:** Executes bytecode either by interpreting it or through JIT compilation.
* **Native Method Interface (JNI):** Allows Java to call native code (written in languages like C or C++).
* **Garbage Collector:** Reclaims memory used by objects that are no longer in use.

**Memory Management in Java**

Memory management in Java is an automatic process largely handled by the JVM, which ensures that memory used by objects is managed efficiently and optimally. Java uses both **heap memory** and **stack memory** for storing data, and the JVM manages these memory regions through its garbage collection mechanism.

**1. Heap Memory:**

* The **heap** is where all Java objects are stored.
* It is divided into two regions: **Young Generation** (for new objects) and **Old Generation** (for long-lived objects).
* Objects are allocated in the young generation, and if they survive garbage collection, they are promoted to the old generation.
* The size of the heap is managed by the JVM and can be adjusted based on application requirements.

**2. Stack Memory:**

* The **stack** is used to store local variables, method calls, and references to objects in the heap.
* Each thread in a Java program has its own stack.
* Stack memory is allocated when methods are invoked and reclaimed once the method finishes execution.

**3. Method Area:**

* The **Method Area** is a shared memory area where class-level information is stored. It includes:
  + **Class definitions** (including bytecode).
  + **Static variables**: These are variables that belong to the class itself, not to instances of the class.
  + **Static methods**: These methods also belong to the class and can be invoked without creating an object of that class.

Static members are stored here because they are part of the class and not of any specific instance. This means that static variables are shared across all instances of the class.

**4. PC Register**:

* This contains the address of the current instruction being executed, but it does not hold static members.

**5. Memory Allocation and Deallocation:**

* The JVM automatically allocates memory for objects during runtime and deallocates memory when objects are no longer referenced (through garbage collection).
* The garbage collector frees up memory by identifying and removing unreachable objects.

**Garbage Collection in Java**

**Garbage Collection (GC)** is a memory management feature in Java that automatically reclaims memory used by objects that are no longer in use. The main goal of garbage collection is to manage memory by identifying objects that are unreachable (i.e., no longer referenced by any part of the program) and releasing the memory they occupy.

**Key Concepts of Garbage Collection:**

1. **Reachability:** An object is considered reachable if there is a chain of references from a root (e.g., static fields, active threads, local variables).
2. **Unreachable Objects:** Objects that are no longer reachable (no references pointing to them) are considered for garbage collection.

**Phases of Garbage Collection:**

1. **Marking Phase:** The garbage collector marks all the reachable objects.
2. **Sweeping Phase:** All unmarked (unreachable) objects are removed, freeing up memory.
3. **Compacting Phase (optional):** The JVM may compact the remaining objects to reduce fragmentation and optimize memory allocation.

**Types of Garbage Collectors in Java:**

* **Serial Garbage Collector:** Works in a single-threaded manner and is suitable for applications with small datasets and limited memory.

java -XX:+UseSerialGC -jar your-application.jar

* **Parallel Garbage Collector:** Uses multiple threads for garbage collection and is suitable for multi-core systems.

java -XX:+UseParallelGC -jar your-application.jar

* **CMS (Concurrent Mark-Sweep) Garbage Collector:** Minimizes pauses during garbage collection by doing most of the work concurrently with application threads.

java -XX:+ ConcMarkSweepGC -jar your-application.jar

* **G1 (Garbage First) Garbage Collector:** Aims to provide low pause times and is designed for applications with large heaps.

Java -XX:+ UseG1GC -jar your-application.jar